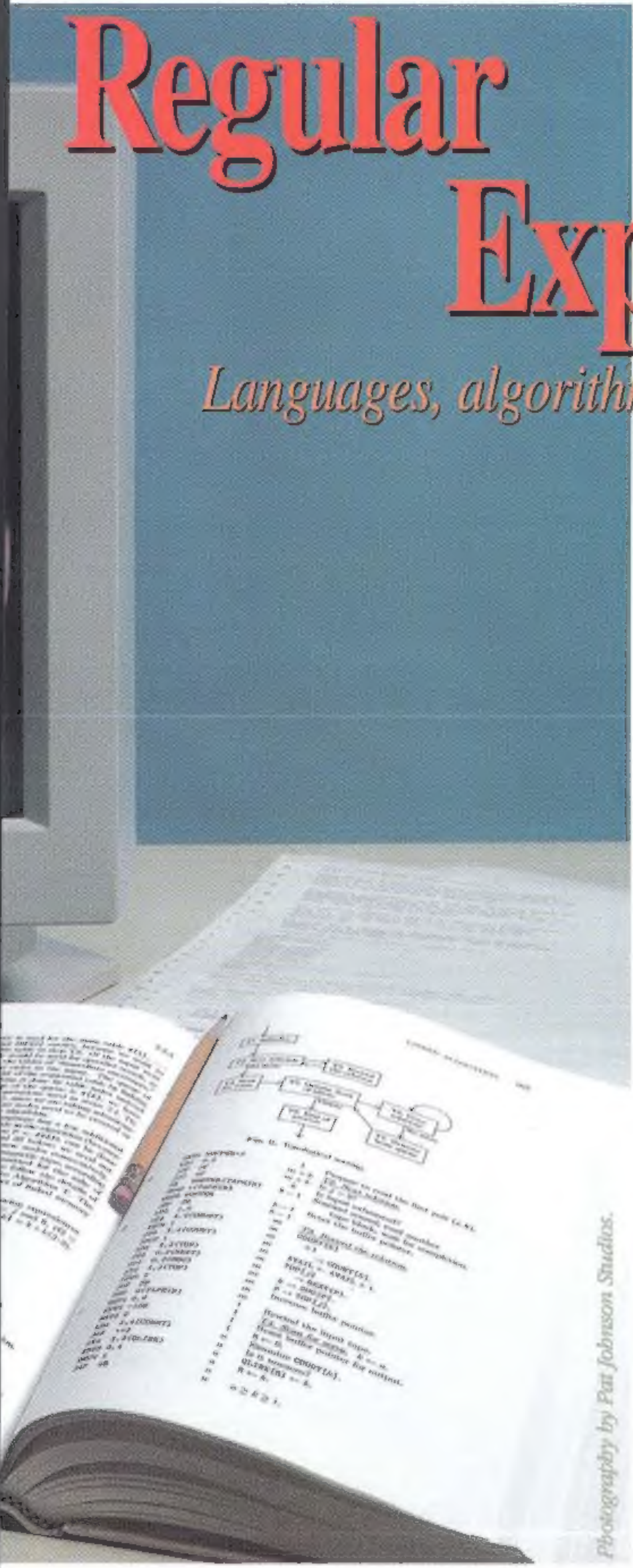
Regular Expressions

Languages, algorithms, and software



Photography by Pat Johnson Studios.

Brian W. Kernighan and Rob Pike

The right programming language can make all the difference in how easy it is to write a program. This is why a programmer's arsenal holds not only general-purpose languages like C and its relatives, but also programmable shells, scripting languages, and lots of application-specific languages.

The power of good notation reaches beyond traditional programming into specialized problem domains. HTML lets us create interactive documents, often using embedded programs in other languages such as JavaScript. PostScript expresses an entire document as a specialized program. Spreadsheets and word processors often include languages to evaluate expressions, access information, and control layout.

Regular expressions are one of the most broadly applicable specialized languages, a compact and expressive notation for describing patterns of text. Regular expressions are algorithmically interesting, easy to implement in their simpler forms, and very useful.

Regular expressions come in several flavors. The so-called "wildcards" used in command-line processors or shells to match patterns of file names are a particularly simple example. Typically, "*" is taken to mean "any string of characters," so a command like

```
del *.exe
```

uses a pattern "*.exe" that matches all files with names that contain any string followed by the literal string ".exe".

Regular expressions pervade UNIX, in editors, tools like grep, and scripting languages like Awk, Perl, and Tcl. Although the variations among different programs may suggest that regular expressions are an ad hoc mechanism, they are, in fact, a language in a strong technical sense— a formal grammar specifies their structure and a precise meaning can be attached to each utterance in the language. Furthermore, the right implementation can run very fast; a combination of theory and engineering practice pays off handsomely.

The Language of Regular Expressions

A regular expression is a sequence of characters that defines a pattern. Most characters in the pattern simply match themselves in a target string, so the regular expression "abc" matches that sequence of three letters wherever it occurs in the target. A few characters are used in patterns as metacharacters

Brian and Rob are researchers at Lucent Technologies's Bell Labs and can be contacted at bwk@bell-labs.com and rob@bell-labs.com, respectively.

to indicate repetition, grouping, or positioning. In POSIX regular expressions, "^" stands for the beginning of a string and "$" for the end, so "^x" matches an "x" only at the beginning of a string, "x$" matches an "x" only at the end, "^x$" matches "x" only if it is the sole character of the string, and "^$" matches the empty string.

The character "." (a period) matches any character, so "x.y" matches "xay," "x2y," and so on, but not "xy" or "xyxy." The regular expression "^.$" matches a string that contains any single character.

A set of characters inside brackets "[]" matches any single one of the enclosed characters; for example, "[0123456789]" matches a single digit. This pattern may be abbreviated "[0–9]."

These building blocks are combined with parentheses for grouping, "|" for alternatives, "*" for zero or more occurrences, "+" for one or more occurrences, and "?" for zero or one occurrences. Finally, "\" is used as a prefix to quote a metacharacter and turn off its special meaning.

These can be combined into remarkably rich patterns. For example, "\.[0–9]+" matches a period followed by one or more digits; "[0–9]+\.?[0–9]*" matches one or more digits followed by an optional period and zero or more further digits; "(\+|–)" matches a plus or a minus ("\+" is a literal plus sign); and "[eE](\+|–)?[0–9]+" matches an "e" or "E" followed by an optional sign and one or more digits. These are combined in the following pattern that matches floating-point numbers:

(\+|–)?([0–9]+\.?[0–9]*|\.[0–9]+)([eE](\+|–)?[0–9]+)?

```
/* match: search for re anywhere in text */
int match(char *re, char *text)
{
    if (re[0] == '^')
        return matchhere(re+1, text);
    do {  /* must look at empty string */
        if (matchhere(re, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
```

Example 1: *The function* match *determines whether a string matches a regular expression.*

```
/* matchhere: search for re at beginning of text */
int matchhere(char *re, char *text)
{
    if (re[0] == '\0')
        return 1;
    if (re[1] == '*')
        return matchstar(re[0], re+2, text);
    if (re[0] == '$' && re[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (re[0]=='.' || re[0]==*text))
        return matchhere(re+1, text+1);
    return 0;
}
```

Example 2: *The recursive function* matchhere *does most of the work.*

```
/* matchstar: search for c*re at beginning of text */
int matchstar(int c, char *re, char *text)
{
    do {  /* a * matches zero or more instances */
        if (matchhere(re, text))
            return 1;
    } while (*text!='\0' && (*text++==c || c=='.'));
    return 0;
}
```

Example 3: *The function* matchstar *is called when the expression begins with a starred character.*

A Regular Expression Search Function

Some systems include a regular expression library, usually called "regex" or "regexp." However, if this is not available, it's easy to implement a modest subset of the full regular expression language. The regular expressions we present here make use of four metacharacters: "^," "$," ".," and "*," with "*" specifying zero or more occurrences of the preceding period or literal character. This provides a large fraction of the power of general regular expressions with a tiny fraction of the implementation complexity. We'll use these functions to implement a small but eminently useful version of grep (available electronically; see "Resource Center," page 5).

In Example 1, the function *match* determines whether a string matches a regular expression. If the regular expression begins with "^" the text must begin with a match of the remainder of the expression. Otherwise, we walk along the text, using *matchhere* to see if the text matches at each position in turn. As soon as we find a match, we're done. Expressions that contain "*" can match the empty string (for example, ".*y" matches "y" among many other things), so we must call *matchhere* even if the text is empty.

In Example 2, the recursive function *matchhere* does most of the work. If the regular expression is empty, we have reached the end and thus have found a match. If the regular expression ends with "$," it matches only if the text is also at the end. If the regular expression begins with a period, it matches any character. Otherwise, the expression begins with a plain character that matches itself in the text. A "^" or "$" that appears in the middle of a regular expression is thus taken as a literal character, not a metacharacter.

Notice that *matchhere* calls itself after matching one character of pattern and string. Thus the depth of recursion can be as much as the length of the pattern.

The one tricky case occurs when the expression begins with a starred character, "x*," for example. Then we call *matchstar* with three arguments—the operand of the star (x), the pattern after the star, and the text; see Example 3. Again, a starred regular expression can match zero characters. The loop checks whether the text matches the remaining expression, trying at each position of the text as long as the first character matches the operand of the star.

Our implementation is admittedly unsophisticated, but it works. And, at fewer than 30 lines of code, it shows that regular expressions don't need advanced techniques to be put to use.

Grep

The pattern-matching program grep, invented by Ken Thompson (the father of UNIX), is a marvelous example of the value of notation. It applies a regular expression to each line of its input files and prints those lines that contain matching strings. This simple specification, plus the power of regular expressions, lets it solve many day-to-day tasks. In the following examples, note that the regular expression used as the argument to grep is different from the wildcard pattern used to specify file names.

- Search for a name in a set of source files: *grep fprintf *.c*
- Search for a phrase in a set of text files: *grep 'regular expression' *.txt*
- Filter output from some other program, for example to print all error messages: *gcc *.c | grep Error.*
- Filter input to some other program, for example to count nonblank lines: *grep . *.cpp | wordcount*

With flags to print line numbers of matched lines, count matches, do case-insensitive matching, select lines that don't match the pattern, and other variations of the basic idea, grep is so

widely used that it has become the classic example of tool-based programming.

Example 4 is the main routine of an implementation of grep that uses *match*. It is conventional that UNIX programs return 0 for success and nonzero values for various failures. Our grep, like the UNIX version, defines success as finding a matching line, so it returns 0 if there were any matches, 1 if there were none, and 2 if an error occurred. These status values can be tested by other programs like a shell.

As Example 5 illustrates, the function *grep* scans a single file, calling *match* on each line. This is mostly straightforward, but there are a couple of subtleties. First, the main routine doesn't quit if it fails to open a file. This is because it's common to say something like

```
% grep herpolhode *.*
```

and find that one of the files in the directory can't be read. It's better for grep to keep going after reporting the problem, rather than to give up and force users to type the file list manually to avoid the problem file. Second, grep prints the matching line and the file name, but suppresses the file name if it is reading standard input or a single file. This may seem an odd design, but it reflects a style of use based on experience. When given only one input, grep's task is usually selection, and the file name would clutter the output. But if it is asked to search through many files, the task is most often to find all occurrences of something, and the file names are helpful. Compare

```
% strings enormous.dll | grep Error:
```

with

```
% grep grammer *.txt
```

Our implementation of *match* returns as soon as it finds a match. For grep, that is a fine default. But, for implementing a substitution (search-and-replace) operator in a text editor, the leftmost longest match is more useful. For example, given the

```
/* grep main: search for re in files */
int main(int argc, char *argv[])
{
    int i, nmatch;
    FILE *f;

    if (argc < 2) {
        fprintf(stderr, "usage: grep pattern [file ...]\n");
        exit(2);
    }
    nmatch = 0;
    if (argc < 3) {
        if (grep(argv[1], stdin, NULL))
            nmatch++;
    } else {
        for (i = 2; i < argc; i++) {
            f = fopen(argv[i], "r");
            if (f == NULL) {
                fprintf(stderr, "grep: can't open %s\n", argv[i]);
                continue;
            }
            if (grep(argv[1], f, argc>3 ? argv[i] : NULL))
                nmatch++;
            fclose(f);
        }
    }
    return nmatch == 0;
}
```

Example 4: *Main routine of an implementation of grep that uses* match. *(Assumes command interpreter expands wild cards in file specification on the command line into lists of file names. UNIX shells exhibit this behavior, although MS-DOS COMMAND.COM does not.)*

```
/* grep: search for re in file */
int grep(char *re, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];

    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(re, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
```

***Example 5:** The function* grep *scans a single file, calling* match *on each line.*

text "aaaaa," the pattern "a*" matches the null string at the beginning of the text, but the user probably intended to match all five characters. To cause *match* to find the leftmost longest string, *matchstar* must be rewritten to be greedy: Rather than looking at each character of the text from left to right, it should skip over the longest string that matches the starred operand, then back up if the rest of the string doesn't match the rest of the pattern. In other words, it should run from right to left. Example 6 is a version of *matchstar* that does leftmost longest matching. This might be the wrong version of *matchstar* for


```
/* matchstar: leftmost longest search for c*re */
int matchstar(int c, char *re, char *text)
{
  char *t;

  for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
    ;
  do {    /* * matches zero or more */
     if (matchhere(re, t))
        return 1;
  } while (t-- > text);
  return 0;
}
```

***Example 6:** A version of* matchstar *that does leftmost longest matching.*

grep, because it does extra work; but for a substitution operator, it is essential.

What Next?

Our grep is competitive with system-supplied versions, regardless of the regular expression. For example, it takes about six seconds to search a 40-MB text file on a 400-MHz Pentium (compiled with Visual C++). Pathological expressions can cause exponential behavior, such as "a*a*a*a*a*b" when given the input "aaaaaaaaac," but the exponential behavior exists in many commercial implementations, too. A more sophisticated matching algorithm can guarantee linear performance by avoiding backtracking when a partial match fails; the UNIX egrep program implements such an algorithm, as do scripting languages.

Full regular expressions would include character classes like "[a-zA-Z]" to match a single alphabetic character; the ability to quote a metacharacter (for example, to search for a literal period); parentheses like "(abc)*" for grouping; and alternatives, where "abc|def" matches "abc" or "def."

The first step is to help *match* by compiling the pattern into a representation that is easier to scan. It is expensive to parse a character class every time we compare it against a character; a precomputed representation based on bit vectors could make character classes much more efficient.

For regular expressions with parentheses and alternatives, the implementation must be more sophisticated. One approach is to compile the regular expression into a parse tree that captures its grammatical structure. This tree is then traversed to create a state machine — a table of states, each of which gives the next state for each possible input character. The string is scanned by the state machine, which reports when it reaches a state corresponding to a match of the pattern. Another approach is similar to what is done in just-in-time compilers: The regular expression is compiled into instructions that will scan the string; the state machine is implicit in the generated instructions.

Further Reading

J.E.F. Friedl's *Mastering Regular Expressions* (O'Reilly & Associates, 1997) is an extensive treatment of the subject. Regular expressions are one of the most important features of some scripting languages; see *The AWK Programming Language* by A.V. Aho, B.W. Kernighan and P.J. Weinberger (Addison-Wesley, 1988) and *Programming Perl*, by Larry Wall, Tom Christiansen, and Randal L. Schwartz (O'Reilly & Associates, 1996).

DDJ